Introduction to Android Programming

Intro

- Shreyansh Barodiya, 3rd year B.Tech.
- Overview of android programming
- Experience during the project

Getting started

- Download android studio from http://developer.android.com/
- Fast emulators - Genymotion - http://www.genymotion.com/
- Careful while installing SDKs (download only the latest 4-5 versions)
- Documentation - https://developer.android.com/training/index.html

Android OS

- The Android operating system is a multi-user Linux system in which each app is a different user.
- By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.
- Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.
- By default, every app runs in its own Linux process. Android starts the process when any of the app's components need to be executed, then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

Sharing data and system services

- It's possible to arrange for two apps to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, apps with the same user ID can also arrange to run in the same Linux process and share the same VM
- An app can request permission to access device data such as the user's contacts, SMS messages, the mountable storage (SD card), camera, Bluetooth, and more. The user has to explicitly grant these permissions.

Project Components

- src – your source code
- gen – auto-generated code (usually just R.java)
- Included libraries
- Resources
    - Drawables (like .png images)
    - Layouts
    - Values (like strings)
- Manifest file

XML files

- Used to define some of the resources
    - Layouts (UI)
    - Strings
- Manifest file
- Shouldn’t usually have to edit it directly, Eclipse/studio can do that for you
- Preferred way of creating UIs
    - Separates the description of the layout from any actual code that controls it
    - Can easily take a UI from one platform to another

R Class

- Auto-generated: you shouldn’t edit it
- Contains IDs of the project resources
- Enforces good software engineering
- Use findViewById and Resources object to get access to the resources
    - Ex. Button b = (Button)findViewById(R.id.button1)
    - Ex. getResources().getString(R.string.hello));

Layouts

- Android studio has a great UI creator
- Generates the XML for you
- Composed of View objects
- Can be specified for portrait and landscape mode
    - Use same file name, so can make completely different UIs for the orientations without modifying any code

Strings

- In res/values
    - strings.xml
- Application wide available strings
- Promotes good software engineering
- UI components made in the UI editor should have text defined in strings.
  xml
- Strings are just one kind of ‘Value’ there are many others

AndroidManifest

- Contains characteristics about your application
- Handling more than one activity
- Need to specify Services and other components too
- Also important to define permissions and external libraries, like Google Maps API

```
<manifest ... >
<uses-feature android:name="android.hardware.camera.any"
              android:required="true" />
  <uses-sdk android:minSdkVersion="7" android:
targetSdkVersion="19" />
  ...
  <application ... >
    <activity android:name="com.example.project.
ComposeEmailActivity">
      <intent-filter>
        <action android:name="android.intent.action.SEND" />
        <data android:type="*/*" />
        <category android:name="android.intent.category.
DEFAULT" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

App Components

- Activities
- Services
- Content Providers
- Broadcast receivers

Activities

- An *activity* represents a single screen with a user interface.
    - For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails.
    - Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others.
- As such, a different app can start any one of these activities (if the email app allows it).
    - For example, a camera app can start the activity in the email app that composes new mail, in order for the user to share a picture.
- An activity is implemented as a subclass of Activity.

Services

- A *service* is a component that runs in the background to perform long-running operations or to perform work for remote processes.
- A service does not provide a user interface.
    - For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity.
- Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it.
- A service is implemented as a subclass of Service

Content Providers

- A *content provider* manages a shared set of app data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your app can access.
- Through the content provider, other apps can query or even modify the data (if the content provider allows it). For example, the Android system provides a content provider that manages the user's contact information. As such, any app with the proper permissions can query part of the content provider (such as ContactsContract.Data) to read and write information about a particular person.
- Content providers are also useful for reading and writing data that is private to your app and not shared.
  - For example, the Note Pad sample app uses a content provider to save notes.
- A content provider is implemented as a subclass of ContentProvider and must implement a standard set of APIs that enable other apps to perform transactions.

Broadcast Receivers

- A *broadcast receiver* is a component that responds to system-wide broadcast announcements.
- Many broadcasts originate from the system—
  - For example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured.
- Apps can also initiate broadcasts—
  - For example, to let other apps know that some data has been downloaded to the device and is available for them to use.
- Although broadcast receivers don't display a user interface, they may create a status bar notification to alert the user when a broadcast event occurs.
- More commonly, though, a broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work.
  - For instance, it might initiate a service to perform some work based on the event.
- A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object.

Activating components

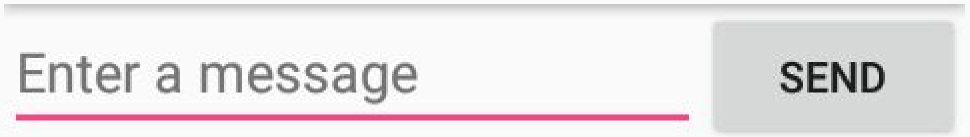
- You can start an activity (or give it something new to do) by passing an Intent to startActivity() or startActivityForResult() (when you want the activity to return a result).
- You can start a service (or give new instructions to an ongoing service) by passing an Intent to startService(). Or you can bind to the service by passing an Intent to bindService().
- You can initiate a broadcast by passing an Intent to methods like sendBroadcast(), sendOrderedBroadcast(), or sendStickyBroadcast().
- You can perform a query to a content provider by calling query() on a ContentResolver.

Creating first app

- src - .java files - brains
- res/layout - design - xml files
- AndroidManifest
- R.java
- Basic views and viewgroups -
  - EditText
  - TextView
  - Button
  - ImageView
  - DatePicker, TimePicker
  - FloatingActionButton

Code snippet (layout resource)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:orientation="horizontal"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
    <EditText android:id="@+id/edit_message"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:hint="@string/edit_message" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/button_send"
        android:onClick="sendMessage"
    />
</LinearLayout>
```



Code snippet (MainActivity.java)

```
public class MainActivity extends AppCompatActivity {
    public final static String EXTRA_MESSAGE = "com.example.myfirstapp.MESSAGE";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** Called when the user clicks the Send button */
    public void sendMessage(View view) {
        Intent intent = new Intent(this, DisplayMessageActivity.class);
        EditText editText = (EditText) findViewById(R.id.edit_message);
        String message = editText.getText().toString();
        intent.putExtra(EXTRA_MESSAGE, message);
        startActivity(intent);
    }
}
```

Code snippet (DisplayMessageActivity.java)

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_display_message);

    Intent intent = getIntent();
    String message = intent.getStringExtra(MainActivity.EXTRA_MESSAGE);
    TextView textView = new TextView(this);
    textView.setTextSize(40);
    textView.setText(message);

    ViewGroup layout = (ViewGroup) findViewById(R.id.activity_display_message);
    layout.addView(textView);
}
```

Android networking

Check the Network Connection

```
public void myClickHandler(View view) {
    ...
    ConnectivityManager connMgr = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();
    if (networkInfo != null && networkInfo.isConnected()) {
        // fetch data
    } else {
        // display error
    }
    ...
}
```

Android networking

Perform Network Operations on Separate Thread

- Uses AsyncTask to create a task away from the main UI thread.
- This task takes a URL string and uses it to create an HttpUrlConnection.
- Once the connection has been established, the AsyncTask downloads the contents of the webpage as an InputStream.
- Finally, the InputStream is converted into a string, which is displayed in the UI by the AsyncTask's onPostExecute method.

```
private class DownloadWebpageTask extends
AsyncTask<String, Void, String> {
        @Override
        protected String doInBackground(String... urls) {
            // params comes from the execute() call: params[0] is
the url.
            try {
                return downloadUrl(urls[0]);
            } catch (IOException e) {
                return "Unable to retrieve web page. URL may be
invalid.";
            }
        }
        // onPostExecute displays the results of the AsyncTask.
        @Override
        protected void onPostExecute(String result) {
            textView.setText(result);
        }
    }
```

Android networking - Volley

- Volley is an HTTP library that makes networking for Android apps easier and most importantly, faster.
- Volley offers following benefits -
  - Automatic scheduling of network requests.
  - Multiple concurrent network connections.
  - Support for request prioritization.
  - Cancellation request API. You can cancel a single request, or you can set blocks or scopes of requests to cancel.
  - Strong ordering that makes it easy to correctly populate your UI with data fetched asynchronously from the network.
- In build.gradle file-

```
dependencies {
        compile fileTree(dir: 'libs', include: ['*.jar'])
        compile 'com.android.support:appcompat-v7:22.2.0'
        compile 'com.android.volley:volley:1.0.0'
}
```

Android networking - Volley

```
StringRequest stringRequest = new StringRequest(Request.Method.POST, url, new Response.Listener<String>() {
                @Override
                public void onResponse(String s) { //do something with the response }
        }, new Response.ErrorListener() {
                @Override
                public void onErrorResponse(VolleyError volleyError) { //do something on receiving the error }
        }){
                @Override
                protected Map<String, String> getParams() throws AuthFailureError {
        Map<String,String> params = new Hashtable<String, String>(); //Creating parameters
        params.put(key, value); //Adding parameters
        ...
        return params;
        }
        };
        RequestQueue requestQueue = Volley.newRequestQueue(this); //Creating a Request Queue
        requestQueue.add(stringRequest); //Adding request to the queue
```

Android Programming Notes

- Android apps have multiple points of entry: no main() method
  - Cannot “sleep” in Android
  - During each entrance, certain Objects may be null
  - Defensive programming is very useful to avoid crashes, e.g.,
    if (!(myObj == null)) { // do something }
- Logging state via android.util.Log throughout app is essential when debugging (finding root causes)
- Better to have “too many” permissions than too few
  - Otherwise, app crashes due to security exceptions!
  - Remove “unnecessary” permissions before releasing app to public

References

- developer.android.com/
- www.vogella.com/
- www.stackoverflow.com
- www.androidhive.info
- www.tutorialspoint.com/
- www.w3schools.com
- docs.google.com/presentation/

Thank You